Project Proposal: Provisioning applications(platform-as-a-service) using containers

Siddhartha Dutta (120040005), Karan Ganju(120050021)

September 20, 2015

1 Problem Description

1.1 Context

Cloud services are currently divided into the following three categories:

1. Infrastructure as a service: Takes physical computer hardware such as servers, storage arrays, and networking and lets users build virtual infrastructure that mimics these resources, but which can be created, reconfigured, resized, and removed within moments. Eg. Amazon EC2, S3.
2. Platform as a service: Provides a platform allowing customers to develop, run and manage Web applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app. Eg. Google App Engine, Heroku, RedHat OpenShift, Cloud Foundary, IBM Bluemix.
3. Software as a service: Built on top of a Platform as a Service solution(could be public or private) and provides software for end-users such as email, word processing or a business CRM which could be charged on a per-user or per-month basis. Eg. Adobe Creative Cloud, Office 365.

PaaS which will be focus of our project is a service required by developers who just want to focus on writing and maintaining their applications without worrying about the infrastructure underneath. They deploy their (usually web) applications on PaaS platforms which take care about providing proper runtime environments and maintaing redundancy, high availability, security and scalability of the applications. Service providers charge customers depending on the guarantees and resources they provide in case its a public service. In private settings quota restrictions can potentially be applied on employees. Cloud virtualization enables developers to be vendor agnostic as they don't need to rely on any particular vendor's API.

PaaS services can be built on top of IaaS or vanilla Linux. One method, when built directly over IaaS whether public(Amazon AWS) or private(OpenStack-based) is to try to run different applications on different VMs. This by its very foundation provides the required isolation. However, its a overkill in terms of resource requirements. Linux has been supporting containers using cgroups even before hardware virtualization based clouds became popular. PaaS services can be provided by making different applications or even different servers of the same application(web, database) run on different containers with all machines running just Linux. This significantly decreases overhead while isolating namespaces but limits flexibility(what about applications that need Windows).

Apart from the above mentioned dilemma, communication between containers, routing, load balancing, authentication, providing a CLI and GUI interface to users, maintaining a registry of images, dealing with node failure, scheduling and placement of containers/ VMs will be some issues that will be addressed by our project. Security to support multiple tenants and netwrok isolation is required. Not everything needs to be done from scratch. Infact, there are opensource libraries/ frameworks that address the mentioned issues. However, finding, understanding and stitching them together is a non-trivial task in itself.

1.2 Problem satement

Building a cloud computing platform as a service(PaaS) enabling developers to deploy applications on demand using containers.

1.3 Deliverables

A front-end(GUI and/ or CLI) for users to:

- state their CPU, RAM, Storage requirments
- deploy their applications(WAR files) or upload their code(could connect to git)

A front-end(GUI and/ or CLI) for administrators to:

- see resource usage in various physical machines
- see resource usage by various users

A backend to:

- measure resource usage of various users
- maintain isolation
- take care of provisioning/ scheduling
- maintain redundancy and availabilty
- authenticate users
- ensure communication between containers of the same user
- maintain security
- manage lifecycles
- balance load
- maintain logs
- check health of applications

Providing runtime environments and frameworks for applications(only a representative subset may be completed depending on the time required for configuration and installation):

- Language environments
  - Java
  - Python
  - PHP
  - Ruby
  - Perl
  - JavaScript
- Databases
  - MongoDB
  - MySQL
  - PostgreSQL
- Java-based servers
  - Apache Tomcat
  - JBoss EAP
- Web application frameworks
  - Node.js(JavaScript)
  - WSGI(Python)

2 Methodology

2.1 Components and Approach Details

We'll run Ubuntu on a set of machines(depending on how many can be made available). DevStack provides a OpenStack implementation for multiple nodes(OpenStack can be tested online using TryStack). We may use JujuCharms which is a service orchestration management tool. Heat is another orchestration tool. Charms can use Docker to different services in different containers. This will enable us to deploy any service MySQL, PostgrSQL etc. on a container of our choosing. Nagios, Boundary, Tivoli can be used to monitor databases. We may end up modifying and using Cynder for block storage, Swift for object storage, Glance to maintain a repository of images, Keystone for authorization and authentication and Horizon for dashboard management. HAProxy can be used for load balancing while Ceilometer can be used to collect measurements. Kubernetes and Mesos together with Marathon and Zookeeper are also used to manage containers. They can be used with/ without OpenStack. Flannel or Open vSwitch can be used for network management.

We'll need to implement(either from scratch or by using above mentioned tools) the following components:

1. Controller
   - Orchestrates all other components.
   - Performs authentication
   - Stores information about the status of other components in the architecture, the users, the deployed applications and available services.

- Exposes a REST interface for accepting requests from the command line tool/ GUI.
- Binds external services like RabbitMQ and MongoDB to the deployed applications.

2. Health Manager
    - Monitors the health of the deployed applications and other components.
    - Corrective actions taken by controller.
    - To find out about the health of an application, it compares the current state of a deployed application with the expected state. The expected state of an application that has been running for some time is derived from its initial state. The expected state of an application is available in the controller's DB.
3. Execution Agent
    - Component responsible for staging and running all applications.
    - Maintains metadata and deploys applications in containers.
    - Handles scaling the applications when the load increases by launching new instances of the application.
    - Monitors the application that it is running, and generates alerts in case of a change of state.
4. Router
    - Takes an incoming request, and forwards it to the appropriate container.
    - Distributes the load among containers like a load balancer.
    - Maintains a routing table which is referred to before making routing decisions.
    - If an application that the router sent a request to has failed, the router retries the request with another instance of the same application.
    - The routing table is updated in real time, based on the status of the containers.
5. Messaging
    - Used by all other components in the architecture for communications.
    - Will use asynchronous messaging semantics for high scalability.
    - Publish/subscribe messaging called NATS will be used for internal communications.
    - Will use RabbitMQ for messaging between applications.
6. Service Abstraction Components
    - Examples of external services are RabbitMQ, MySQL, mongoDB and Redis.
    - To interface with them, a Service Provisioning Agent can be used. There will be one Service Provisioning Agent for each external service.
    - The Service Gateway is the interface for the Controller to provision external services and to track the status of those services.
    - Different services can be tied to each other.
    - This module will most likely(I'm yet to explore it completely) make calls to JujuCharms.

2.2 Ordering

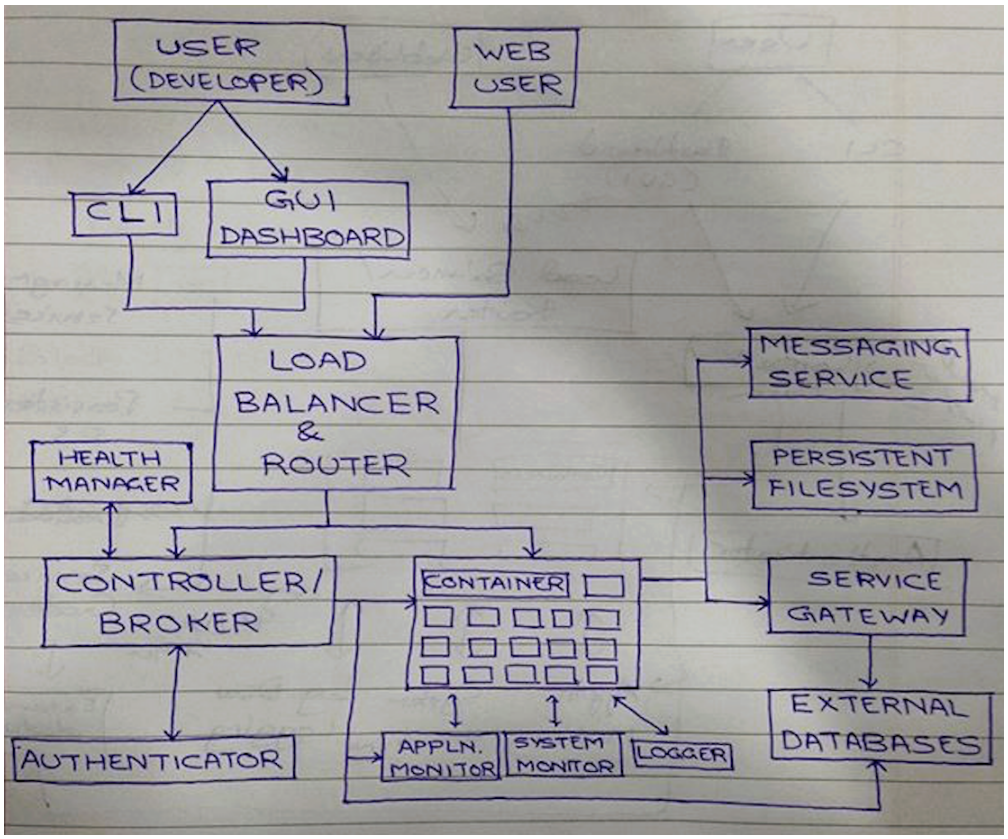

Figure 1: Ordering

3 Evaluation

3.1 Questions

1. Performance:
   (a) Ability to honor SLAs or resource guarantees under heavy load
   (b) Efficiency of resource pooling
   (c) Rapid Elasticity? Speed of provisioning(during first instantiation as well as during sudden increase in demand)
   (d) Inter container communication speed/ delay/ latency
   (e) Difference in time required to handle HTTP requests when using containers in our cloud and in case they were running on a single machine
   (f) Speed up due to load balancer
   (g) Overhead of applications that help in the running of the cloud(i.e. due to the components mentioned above)

(h) Mean time to respond after a failure

2. Correctness: We plan to use robust protocols like TCP for communication so there's no question of errors in messages creeping in. However, it could be the case that if a task is distributed amongst containers and the result from all of them is summed up to give the final output incorrect results may come up if there are problems in the way tasks are split up into containers and then retrieved or because of errors in synchronization. Whether or not such problems can come up will depend on our design which is yet to fully complete.

3. Tradeoffs: There's an inherent tradeoff we make during provisioning. Whether or not to keep free memory to allow running applications to scale up faster on the same machine or cram as many containers on the same physical machines as possible to make best use of running machines.

3.2 Setup

Functions/ modules will be inserted at appropriate places to collect statistics about the fore-mentioned parameters. For example, to measure the delay in servicing HTTP requests, we will track time in the controller to see how much time was spent between the request from a particular IP arrived and when it was serviced(ie. responses were sent). Test java applications will be written and performance will be checked. Nodes will be deliberately failed in order to see the effectiveness of replication and check for disruption. Automated requests will be made for the web applications to see the effectiveness of horizontal scaling.

4 Timeline

Week 1	Enrich and complete the described design and arrange machines
Week 2	Install linux, docker, openstack and other tools on all machines
Week 3&4	Implement the controller, execution agent, messaging, router and service abstraction components
Week 5	Make the front end webpage/ dashboard, command line tool, install frameworks for the environments we want to provide to applications
Week 6	Carry out evaluation